

On Effectiveness of a Message-Driven Confidence-Driven Protocol for Guarded Software Upgrading*

Ann Tai Kam Tso Leon Alkalai Savio Chau William Sanders
IA Tech, Inc. Jet Propulsion Laboratory University of Illinois
Los Angeles, CA 90024 Pasadena, CA 91109 Urbana, IL 61801

Abstract

In order to accomplish dependable onboard evolution, we develop a methodology which is called guarded software upgrading (GSU). The core of the methodology is a low-cost error containment and recovery protocol that escorts an upgraded software component through onboard validation and guarded operation, safeguarding mission functions. The message-driven confidence-driven (MDCD) nature of the protocol eliminates the need for costly process coordination or atomic action, yet guaranteeing the system to reach a consistent global state upon the completion of the rollback or roll-forward actions carried out by individual processes during error recovery. Aimed at validating the effectiveness of the MDCD protocol with respect to its ability, in a realistic, non-ideal execution environment, to enhance system reliability when a software component undergoes onboard upgrading, we conduct a stochastic activity network model based analysis. The results confirm the effectiveness of the protocol as originally surmised. Moreover, the model-based analysis provides to us useful insights about the system behavior resulting from the use of the protocol under various conditions in its execution environment, facilitating effective utility of the protocol.

Keywords: Guarded software upgrading, error containment and recovery, checkpointing, stochastic activity networks, reliability improvement

Corresponding Author: Ann T. Tai, a.t.tai@ieee.org

*The work reported in this paper was supported in part by Small Business Innovation Research (SBIR) Contract NAS3-99125 from Jet Propulsion Laboratory, National Aeronautics and Space Administration.

1 Introduction

The onboard computing systems for NASA's future deep-space applications require to have the ability to accomplish performance and dependability enhancement during a long-life mission [1]. This capability is referred to as *evolvability*. Concepts related to evolvability include hardware reconfigurability and software upgradability. A challenge that arises from onboard software upgrade is to guard the system against performance loss caused by residual design faults introduced by the modification of a spacecraft/science function. Besides the previous lessons on how unprotected software upgrades caused severe damages to space missions (see [2, 3], for example), a strong testimony emerged from MCI WorldCom's recent 10-day frame relay outage [4]. The outage began August 5, 1999, four weeks after an upgrade to a new switching software to allow the network to handle increased traffic. The incident affected about 15% of MCI WorldCom's network and 30% of its customers who rely on the high-speed frame relay.

Although researchers have been investigating into dependable system upgrade for critical applications [5, 6], the proposed solutions, to our best knowledge, all require special effort for developing dedicated system resource redundancy. Due to the severe constraints on cost, mass and power consumption of the spacecraft, NASA's deep-space applications would not be able to directly benefit from those solutions. Moreover, the new-generation onboard computing systems such as the X2000 [1] which has being developed at NASA/JPL employ distributed architectures. Accordingly, error contamination among interacting processes, which received little attention from the prior work concerning dependable system upgrade, is one of our major concerns. With the above motivation, we develop a methodology called guarded software upgrading (GSU) [7]. The methodology is based on a two-stage approach: The first stage is called *onboard validation* stage during which we attempt to establish high confidence in the new version, through onboard test runs under the real avionics system and environment conditions; whereas the second stage is called *guarded operation* stage during which we allow the new version to actually service the mission under the escort of the old version. To ensuring low development cost, we exploit inherent system resource redundancies as the fault tolerance means. Specifically, we let an old version, in which we have high confidence due to its long onboard execution time, to escort the new version through onboard validation and guarded operation; we also make use of the processor that otherwise would be idle during a non-critical mission phase during which onboard software upgrade takes place, allowing concurrent execution of the new and old versions of the application software component aimed for upgrading. To reduce performance cost, we devise an error containment and recovery protocol by utilizing the pertinent features of our application. In particular,

we discriminate i) between internal and external messages in terms of their criticality to the mission, and ii) between the individual software components with respect to our confidence in their reliability. As a result, the protocol is message-driven and confidence-driven (MDCD), requiring no costly process coordination or atomic action. The MDCD protocol permits the decisions on whether to take a checkpoint upon interprocess communication and whether to rollback or roll forward during recovery to be made locally by individual processes, enabling cost-effective checkpointing and cascading-rollback free error recovery. In this paper, we focus on analyzing the effectiveness of the protocol during the guarded operation stage.

To account for potential process state contamination (due to the errors in the new version) and messages validity, we adapt the notion of “global state consistency” from the literatures concerning rollback recovery of hardware faults [8, 9]. Based on the adapted notion, we have developed theorems and conducted formal proofs to show that the MDCD protocol guarantees that the system will reach a consistent global state upon the completion of the rollback or roll-forward actions carried out by individual processes during error recovery [10, 11]. Global state consistency is the most fundamental criterion for a correct recovery and assures the system to be failure free if the MDCD protocol is run in an *ideal execution environment* where 1) the “old” software components which are viewed as the high-confidence components by the protocol are truly faultless, 2) error conditions in a process state will be definitely manifested in the messages sent by the corresponding process, and 3) the error detection mechanism has a perfect coverage. As any reliability enhancement schemes, the realistic goal of the MDCD protocol is to significantly reduce system failure probability rather than assuring the system to be failure free, due to that the ideal execution environment never exists in real life. Accordingly, we are motivated to validate the protocol’s effectiveness in terms of reliability improvement when the criteria for the ideal execution environment are not satisfied, through probabilistic modeling. To realize the goal requires a model to capture numerous interdependencies among system attributes. Accordingly, we choose to apply stochastic activity networks (SANs) [12, 13] for the model-based analysis due to SANs’ capability of explicitly representing dependencies among system attributes. The results confirm the effectiveness of the MDCD protocol and provide to us useful insights about the system behavior in a non-ideal environment. Moreover, the study demonstrates that a model-based analysis will enable us to estimate the extents to which we can relax the the criteria for the ideal execution environment, facilitating effective utility of the protocol.

The remainder of the paper is organized as follows. Section 2 provides background information. Section 3 describes the MDCD protocol for guarded software upgrading, followed by Section 4 which presents a SAN model based reliability analysis that validates the effectiveness of the protocol. The concluding remark highlights the significance of this effort.

2 Background: GSU Framework

The GSU framework is based on the Baseline X2000 First Delivery Architecture that comprises three high-performance computing nodes (each of which has a 128-Mbyte local DRAM), a group of subsystem micro-controllers, and a variety of devices, all connected by the fault-tolerant bus network that complies with the commercial interface standard IEEE 1394 [14]. Further, the DMA (direct memory access) engine that is built into the IEEE 1394 bus interface is responsible for transferring data (messages) directly from the bus into either the non-volatile memory on the PCI bus or the processor local memory (where the message buffer resides).

Since a software upgrade is normally conducted during a less critical mission phase when the spacecraft and science functions do not require a full computation power, only two processes corresponding two different application software components are supposed to concurrently run and interact with each other. To exploit inherent system resource redundancies, we let the old version in which we have high confidence due to its long onboard execution time escort the new version software component through onboard validation and guarded operation. Further, we make use of the processor that otherwise would be idle to enable the three processes (i.e., the two corresponding to the new and old versions, and the process corresponding to the second application software component) to execute concurrently. To aid the description, we introduce the following notation:

- P_1^{new} The process corresponding to the new version of an application software component.
- P_1^{old} The process corresponding to the old version of the application software component.
- P_2 The process corresponding to another application software component (which is not undergoing upgrade).

Figure 1 illustrates the two-stage approach. As shown in Figure 1(a), during the onboard validation stage, the outgoing messages of the shadow process P_1^{new} are suppressed but selectively logged (as shown by the dashed lines with arrows), while P_1^{new} receives the same incoming messages as the active process P_1^{old} does (as shown by the solid lines with arrows). Thus, P_1^{new} and P_1^{old} can perform the same computation based on identical input data.

By maintaining an onboard error log that can be downloaded to the ground to facilitate statistical modeling and heuristic trend analysis, onboard validation facilitates the decisions on whether and when to permit P_1^{new} to enter mission operation. If onboard validation completes successfully, then P_1^{new} and P_1^{old} switch their roles and enter the guarded operation

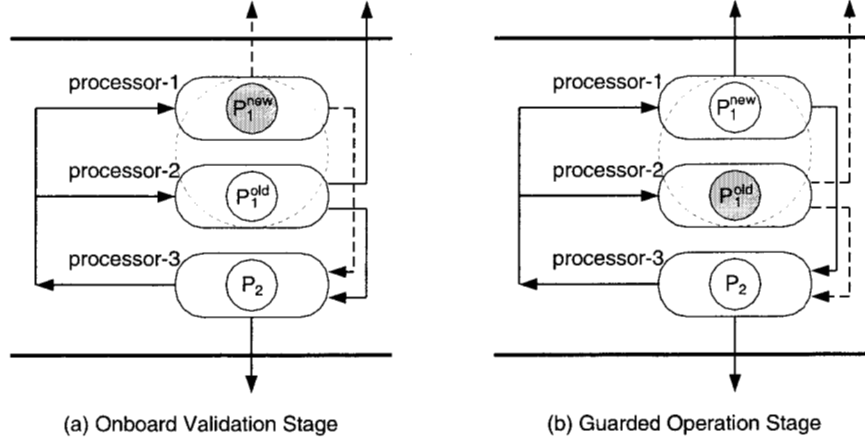


Figure 1: Two-Stage Approach to Guarded Software Upgrading

stage. In order to minimize the impact and risk on mission operation, onboard software upgrading is usually carried out in an incremental manner. In particular, most upgrades involve only a single software component at a time. As a result, the interaction patterns (message types and ordering) among the processes will remain the same after an upgrade. Accordingly, as indicated by Figure 1(b), during the guarded operation, P_1^{new} actually influences the external world and interacts with process P_2 , while the messages of P_1^{old} that convey its computation results to P_2 or external subsystems are now suppressed and logged. Should an error of P_1^{new} be detected, P_1^{old} will take over P_1^{new} 's active role and the system will resume its normal mode. The guarded operation is enabled by an error containment and recovery protocol that is described in the next section.

3 MDCD Protocol

3.1 Basic Assumptions

The following are our assumptions based on which we devise the error containment and recovery protocol:

- A1) The old version of a software component usually has a sufficiently long onboard execution time and thus can be considered significantly more reliable than the upgraded version newly installed through uploading.
- A2) An erroneous state of a process is likely to affect the correctness of its outgoing messages, while an erroneous message received by an application software component will result in process state contamination.

- A3) The error detection mechanism, acceptance test (AT), has a high coverage (the conditional probability that the testing mechanism rejects a computation result given that this result is erroneous).

A1 indicates that the likelihood that an error condition occurs in the old version of an application software component can be considered negligible, implying P_1^{old} and P_2 will not be treated as possible sources of process state contamination by the protocol. A2 implies that if an outgoing message is validated by AT, then the process state of the sender process and all the messages sent or received prior to performing the AT can be considered *non-contaminated* and *valid*, respectively. Whereas A3 suggests that the event that an erroneous command is released to devices is unlikely to occur.

3.2 Protocol Description

A major difficulty in error recovery for embedded systems is that we are unable to rollback the effect of a computation error after it propagates to the devices. Since error propagation in a distributed system is in general caused by message passing, the error containment and recovery protocol we devise is message driven in the sense that,

- 1) *Checkpointing* is performed upon message passing or an event triggered by message passing, and
- 2) *Acceptance test (AT)* is invoked when a process attempts to send a message to a subsystem external to the computing node (e.g., a device).

We call the messages sent to the subsystems external to a computing node and the messages between processes *external messages* and *internal messages*, respectively. In embedded systems, external messages are significantly more critical than internal messages because i) they directly influence the mission operation and functions, and ii) their adverse effects can not be reversed through rollback. Hence, for reducing performance cost, AT is only applied to validate the external messages from the processes that are *potentially contaminated* (see below for the definition). Further, P_1^{old} does not perform AT because its external messages will not be released to devices until after error recovery. On the other hand, when P_1^{new} or P_2 passes an AT successfully, it sends a notification message to P_1^{old} to let it update its knowledge about the validity of process state and messages.

To eliminate the need for the costly process coordination or atomic action, we enforce the following rule (which is indeed the necessary and sufficient condition for checkpointing) to facilitate error containment and recovery efficiency:

We save the state of a process via checkpointing if and only if the process is under the following situation: Immediately before its process state becomes potentially contaminated or right after its process state gets validated by acceptance test.

By “a potentially contaminated process state,” we mean 1) the process state of P_1^{new} in which we have not yet established enough confidence, or 2) a process state that reflects the receipt of a not-yet-validated message that is sent by a process when its process state is potentially contaminated. Figure 2 illustrates the above concepts. The horizontal lines in the figure represent the software executions along the time horizon. Each of the shaded regions represents the execution interval during which the state of the corresponding process is potentially contaminated. In the diagram, checkpoints B_k , A_j and B_{k+2} are established immediately before a process state becomes potentially contaminated (we call them *Type-1* checkpoints), while B_{k+1} , A_{j+1} , and B_{k+3} are established right after a process state gets validated (we call them *Type-2* checkpoints). While all these checkpoint establishments are triggered by the events of potential process state contamination and process state validation, the triggering events themselves are induced by message passing. Therefore, checkpointing is message driven in the protocol. Nonetheless, message passing is not the sufficient condition for a process to establish a checkpoint. As implied by the necessary and sufficient condition for checkpointing stated above, *message passing will not trigger a process to establish a checkpoint unless the message passing event alters our confidence in the process state(s)*, that is, turning a potentially contaminated process state into a validated state or vice versa. Therefore, the protocol is both message driven and confidence driven. The detailed error containment and recovery algorithms that constitute the MDCD protocol are shown in Appendix A. Note that P_1^{old} and P_2 will update their knowledges about potential process state contamination right after a Type-1 or Type-2 checkpoint establishment (e.g., each process will set its dirty bit to 1 and 0, respectively).

Error recovery actions are also message driven and confidence driven in the sense that the AT-based error detection are triggered by the event that P_1^{new} or P_2 attempts to send an external message. Upon the detection of an error, P_1^{old} will take over P_1^{new} ’s active role and resume normal computation with P_2 (the MDCD protocol will go on leave accordingly). By checking their knowledge about process state contamination locally, both P_1^{old} and P_2 are able to make their decisions on rollback or roll forward in a straight forward manner. Accordingly, there are three possible scenarios in error recovery¹:

Scenario 1: Both P_1^{old} and P_2 rollback to their most recent checkpoints.

¹Note that the scenario in which P_1^{old} rolls back and P_2 rolls forward will never happen since P_1^{old} will get contaminated only through P_2 .

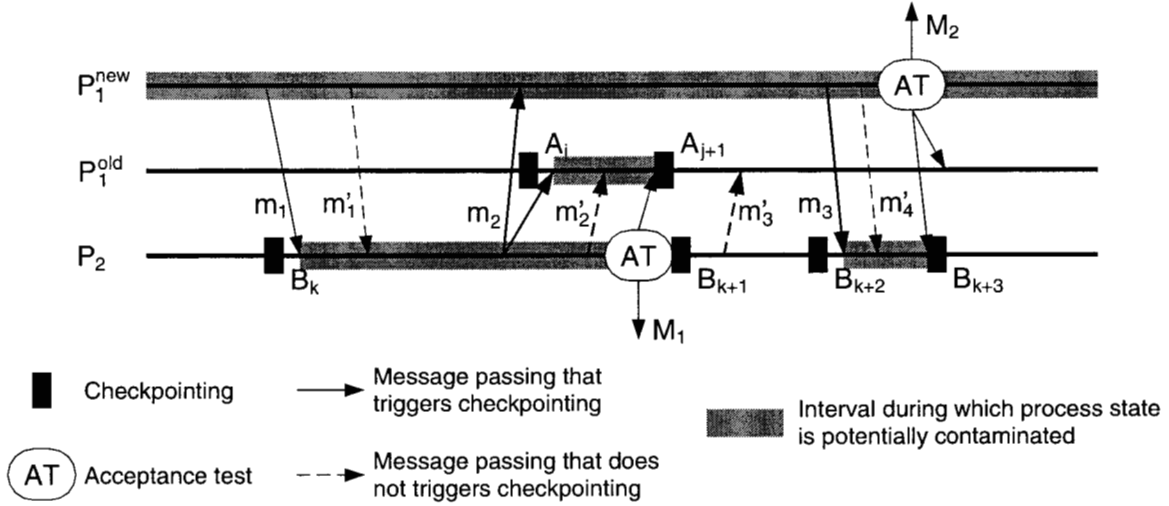


Figure 2: Message-Driven Confidence-Driven Checkpoint Establishment

Scenario 2: Both P_1^{old} and P_2 roll forward.

Scenario 3: P_2 rolls back to its most recent checkpoint while P_1^{old} rolls forward.

Note that our message-driven confidence-driven strategy is adapted from the checkpointing techniques for hardware error recovery [8]. Nonetheless, checkpointing techniques for hardware error recovery concern solely the consistency between process states for assuring correct recovery from hardware faults; as our objective is to mitigate the effect of residual faults in an upgraded software component, our particular concern is the consistency among the views of different processes on process states integrity, especially on the *valid messages* (see Section 3.1) reflected in the process states. Accordingly, we adapt the terminologies and definitions in [8, 9] as follows. A *global state* comprises the states of individual processes, including messages between the processes and *information concerning their verified correctness*. A valid checkpointing mechanism must assure that it is always possible for the error recovery mechanism to bring the system into a global state that satisfies the following two properties:

Consistency If m is reflected in the global state as a valid message received by a process, then m must also be reflected in the global state as a valid message sent by the sender process.

Recoverability If m is reflected in the global state as a valid message sent by a process, then m must also be reflected in the global state as a valid message received by the receiving process(es) or the error recovery algorithm must be able to restore the message m .

When two or more process states (or checkpoints reflecting process states) comprise a global state that satisfies the consistency property, we say that these process states are *globally consistent*, or say, they comprise a *consistent global state*. Based on the above concepts, we derive Theorem 1, Corollaries 1 and 2 as presented below (the formal proofs can be found in [10, 11]), which claim that the recovery decisions made locally by the individual processes satisfy the global state consistency property.

Theorem 1 *The most recent checkpoints of P_1^{old} and P_2 are always globally consistent.*

Corollary 1 *The process states of P_1^{old} and P_2 at time t that are not potentially contaminated are globally consistent.*

Corollary 2 *If at time t the process state of P_2 is potentially contaminated but that of P_1^{old} is not, then the process state of P_1^{old} at time t and the process state of P_2 reflected in its most recent checkpoint (relative to t) are globally consistent.*

As to recoverability, it is ultimately assured by the message log of P_1^{old} and two key entities, namely, `msg_count` and VR_1^{new} (see Appendix A). We omit further details since they are out of the scope of this paper.

4 Analysis of Effectiveness

4.1 Motivation

The theorems presented in the last section imply that the MDCD protocol will assure the system to reach a consistent global state upon the completion of the rollback or roll-forward actions carried out by individual processes during error recovery. As claimed in Section 1, the global state consistency will further guarantee the system to be failure free if the MDCD protocol is run in an ideal execution environment. By “ideal execution environment,” we mean an execution environment for the protocol that satisfies the following criteria:

- C1) P_1^{old} and P_2 are perfectly reliable.
- C2) Error conditions in a process state will be definitely manifested in the messages sent by the corresponding process.
- C3) Each AT has a perfect coverage.

As the realistic goal of the MDCD protocol is to significantly reduce the probability of system failure rather than assuring the system to be failure free, the protocol is anticipated to be effective in a non-ideal execution environment. Accordingly, the motivation of the model-based reliability analysis presented below is to validate the effectiveness of the protocol when it is run in an environment where C1, C2 and C3 are relaxed. Before we proceed to describe the SAN models, we explain the impacts from relaxing these criteria on system failure behavior as follows. Clearly, an imperfect coverage of AT may cause an erroneous external message to go undetected and thus lead to an immediate system failure. And a fault in P_1^{old} or P_2 may result in an undetected external erroneous message after error recovery that brings the system back to its normal computation mode in which AT is no longer performed to validate external messages. Whereas if error manifestation in messages is indeterministic, an erroneous process state may left behind error recovery, which in turn, could eventually lead to system failure. Consider the scenario illustrated in Figure 2, if a residual fault in P_1^{new} causes an error condition before P_1^{new} sends P_2 message m_1 and the error condition is subsequently manifested in m_1 , then P_2 gets contaminated. However, if the error condition in the contaminated P_2 is not manifested in M_1 , the external message P_2 subsequently intends to send, then the corresponding AT will be unable to detect the state contamination. It follows that the contaminated process state will be saved in checkpoints B_{k+1} and B_{k+2} . Consequently, if P_1^{new} fails AT when attempting to send M_2 , P_2 will rollback to B_{k+2} that contains dormant error conditions; and P_1^{old} will simply roll forward because its process state is considered non-contaminated by the protocol (regardless P_1^{old} may gets contaminated through messages m'_2 and m'_3 from P_2). Although the process state of P_2 reflected in B_{k+2} and the process state of P_1^{old} upon recovery are globally consistent, the dormant error conditions may cause the system to fail eventually.

Note that criteria C1, C2 and C3 for the ideal execution environment of the MDCD protocol are similar to but stronger than assumptions A1, A2 and A3 based which we devise the protocol (Section 3.1), respectively. In order to validate the effectiveness of the protocol with respect to reliability improvement under realistic, non-ideal conditions, we carry out probabilistic modeling by relaxing the criteria for the ideal environment as described below.

4.2 SAN Models

Stochastic activity network, a variant of stochastic Petri net (SPN), is first introduced in [12] and currently employed in evaluation tools such as UltraSAN [13]. Through the use of additional primitives such as *cases*, *input gates* and *output gates*, SANs have a relatively rich syntax for the purpose of specifying a complex stochastic process. Specifically, cases permit

an expression of uncertainty about the marking that results from the “completion of an activity” (analogous to the “firing of an SPN transition”), specified by a discrete probability distribution over the cases of that activity. Moreover, the values of this distribution can depend on the marking of the network. In other words, SANs permit an explicit specification of spatial as well as temporal uncertainty. Input and output gates associated with an activity describe, respectively, how that activity is enabled and how its completion affects the subsequent marking of the network. More precisely, input gates permit a functional specification of the enabling predicate and marking updates; output gates specify how the markings of the output places are altered when the activity completes.

Recall that the MDCD protocol is intended to achieve error containment and recovery efficiency by discriminating between the individual software components with respect to our confidence in their reliability. Accordingly, the behavior of the three processes, namely, P_1^{new} , P_1^{old} and P_2 , resulting from the protocol exhibit little symmetry, which could lead to a complex model. However, by exploiting SANs’ marking dependent specification capability, we obtain a rather concise SAN model that captures all the relevant details of the system behavior resulting from the MDCD protocol, as shown in Figure 3.

The SAN representation can be viewed consisting of three parts. The major components of the left part are the timed activities $P1\text{Nec}$, $P10\text{ec}$ and $P2\text{ec}$ which represent the error condition occurrence in P_1^{new} , P_1^{old} and P_2 , respectively. By assigning a non-zero (Poisson) failure rate to each of the timed activities, we relax criterion C1. Recall that P_1^{old} and P_2 are regarded as high-confidence components in the system by the MDCD protocol, meaning that the error conditions in P_1^{old} and P_2 caused by their own faults will be neglected by the error containment and recovery mechanisms of the protocol. This necessitates different representations of error conditions caused by the faults in differing processes. Therefore, while the output gate $P1\text{Nerr}$ sets the marking of the output place $P1\text{Nctn}$ to one upon the completion of $P1\text{Nec}$, the output gates $P10\text{err}$ and $P2\text{err}$ will result in two tokens in $P10\text{ctn}$ and $P2\text{ctn}$, upon the completion of $P10\text{ec}$ and $P2\text{ec}$, respectively.

The middle part of the SAN representation comprises the timed activities $P1\text{Nmsg}$, $P10\text{msg}$ and $P2\text{msg}$. These three activities play important roles in representing the interdependencies among the processes in terms of error contamination. By specifying marking-dependent probability distributions over the cases of these timed activities, uncertainty about the manifestation of error conditions in a contaminated process state in the messages generated by the process is explicitly represented, which enables us to relax criterion C2. As shown in Table 1, the possible combinations of the characteristics of an outgoing message from P_1^{new} are enumerated by the cases of the activity $P1\text{Nmsg}$. Specifically, each message is first characterized by the external and internal message types probabilistically. And if the message is

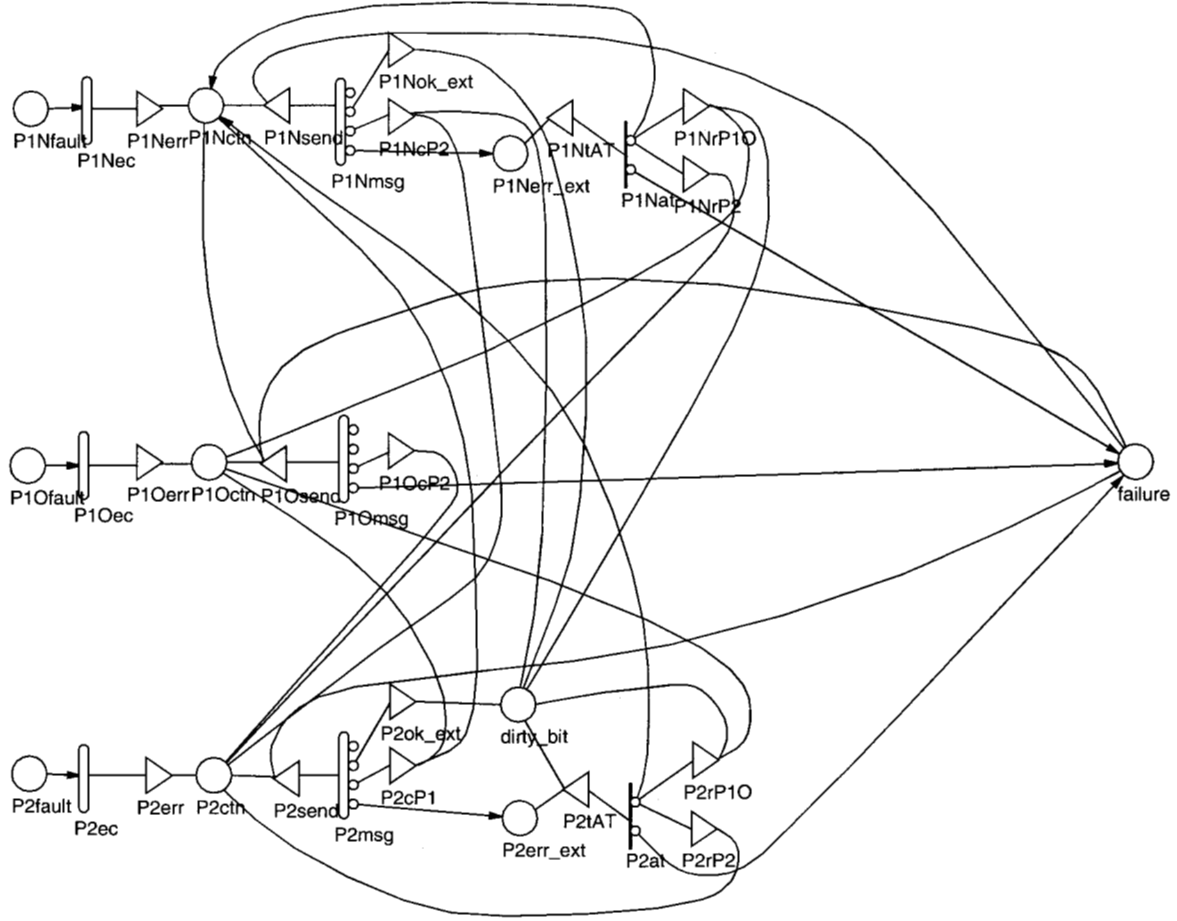


Figure 3: SAN Model for the MDCD Protocol

generated when the process is in an erroneous state, which will be indicated by the marking of the input place $P1Nctn$, then the message will be further characterized probabilistically with respect to whether being affected by the error conditions in the process state. However, for the circumstance where the process state of P_1^{new} is not erroneous, which will be indicated by the empty marking of $P1Nctn$, the above uncertainty is irrelevant. Accordingly, by assigning a zero probability to each, cases 3 and 4 which represent erroneous internal and external messages, respectively, become degenerate. The timed activities $P2msg$ and $P1Omsg$ are specified in a similar manner. However, the output functions of $P1Omsg$ are simpler due to that the messages of P_1^{old} are suppressed prior to error recovery and thus will not influence the correctness of other processes.

Message-passing caused process state contaminations are represented by the output gates $P1NcP2$, $P1OcP2$ and $P2cP1$ which are connected to the cases (of the timed activities $P1Nmsg$, $P1Omsg$ and $P2msg$, respectively) representing erroneous internal messages. Because $C1$ is relaxed in this model whereas P_1^{old} and P_2 are not considered as the sources of process state

Table 1: Case Probabilities for Timed Activity P1Nmsg

| Activity | Case | Probability |
|----------|------|---|
| P1Nmsg | 1 | <pre> if (MARK(P1Nctn)==0) /* non-contaminated internal msg from a non-contaminated state */ return(1-GLOBAL.D(prob_ext)); /* non-contaminated internal msg from a contaminated state */ else return((1-GLOBAL.D(prob_ext))*(1-GLOBAL.D(prob_s2m))); </pre> |
| | 2 | <pre> if (MARK(P1Nctn)==0) /* non-contaminated external msg from a non-contaminated state */ return(GLOBAL.D(prob_ext)); /* non-contaminated external msg from a contaminated state */ else return(GLOBAL.D(prob_ext)*(1-GLOBAL.D(prob_s2m))); </pre> |
| | 3 | <pre> if (MARK(P1Nctn)==0) /* contaminated internal msg from a non-contaminated state */ return(ZERO); /* contaminated internal msg from a contaminated state */ else return((1-GLOBAL.D(prob_ext))*GLOBAL.D(prob_s2m)); </pre> |
| | 4 | <pre> if (MARK(P1Nctn)==0) /* contaminated external msg from a non-contaminated state */ return(ZERO); /* contaminated external msg from a contaminated state */ else return(GLOBAL.D(prob_ext)*GLOBAL.D(prob_s2m)); </pre> |

contamination by the MDCD protocol, we again need to make the representations of the resulting erroneous states discriminable with respect to the source of the contamination. Accordingly, as shown in Table 2, each of the output functions of P1NcP2, P10cP2 and P2cP1 first examines whether the “target” process state (P_1^{old} or P_2) is already contaminated by its own error and if so, the marking that indicates the own-error caused process state contamination will be preserved.

Table 2: Output Gate Definitions for Modeling Error Contamination

| Gate | Definition |
|--------|--|
| P1NcP2 | <pre> if (MARK(P2ctn) != 2) MARK(P2ctn) = 1; MARK(dirty_bit) = 1; </pre> |
| P10cP2 | <pre> if (MARK(P2ctn) != 2) MARK(P2ctn) = 1; </pre> |
| P2cP1 | <pre> if (MARK(P10ctn) != 2) MARK(P10ctn) = 1; if (MARK(P1Nctn) == 0) MARK(P1Nctn) = 1; </pre> |

The output gates P1Nok_ext and P2ok_ext are connected to, respectively, the cases of P1Nmsg and P2msg that represent successful external message sending. The output functions

of these two gates are just resetting the marking of the place `dirty_bit` (to zero), which implies that the process state of P_2 is validated through a successful AT. Although P_2 will not perform AT for its external messages if its process state is not considered contaminated according to the MDCD protocol, a separate representation for this scenario is not required. This is because the marking of `dirty_bit` would be zero before the completion of the activity `P2msg` for this scenario and thus resetting will have no effect. This in turn, implicitly represents the scenario that P_2 sends a correct external message (when its process state is considered not contaminated) without performing AT.

The right part of the SAN model consists of instantaneous activities `P1Nat` and `P2at`. The first and second cases (in a top-down order) of each of the activities, respectively, represent the scenarios where an erroneous external message that is detected by AT triggers error recovery and an undetected erroneous external message causes system failure. For the first case, the corresponding output gates will 1) set the marking of the place `dirty_bit` to zero, and 2) set the markings of the places `P10ctn` and `P2ctn` to zero if the markings prior to the completion of `P2at` are equal to one, implying the rollback recovery brings the processes to the non-contaminated states saved in their most recent checkpoints. Meanwhile, the marking of `P1Nctn` will be set to two, indicating that P_1^{new} stops execution upon error recovery. On the other hand, if the marking of `P10ctn` or `P2ctn` is equal to two, which implies that the state contamination is caused by an error of P_1^{old} or P_2 itself, respectively, the marking will not be altered by the output gates representing recovery actions. This is because the MDCD protocol does not consider that P_1^{old} and P_2 are the potential sources of error contamination and thus will not be able to assure the global state after recovery to be free of the error conditions caused by P_1^{old} and P_2 themselves. The second cases of the activities `P10at` and `P2at` are self-explanatory, i.e., the outcome (an undetected erroneous external message) will simply set the marking of the place `failure` to one. The case probability specification of `P2at` as shown in Table 3 is also marking dependent. This is necessary because P_2 does not perform AT for its external messages 1) after error recovery, or 2) when its process state is considered not contaminated. It is worth to note that the marking dependent case probability specification indeed treats the above two scenarios as a limiting case in which the coverage of AT is zero.

In order to evaluate the effectiveness of the MDCD protocol in terms of reliability improvement. We also construct a SAN model which represents the “baseline system” where the MDCD protocol is not applied. The model is shown in Figure 4, which is quite simple and self-explanatory.

Table 3: Case Probabilities for instantaneous Activity P2at

| Activity | Case | Probability |
|----------|------|--|
| P2at | 1 | <pre> if (MARK(P1Nctn) == 1 && MARK(dirty_bit) == 1) /* AT is performed before recovery */ return(GLOBALD(at_coverage)); /* AT is not performed after recovery or when dirty_bit is zero */ return(ZERO); </pre> |
| | 2 | <pre> if (MARK(P1Nctn) == 1 && MARK(dirty_bit) == 1) /* AT is performed before recovery */ return(1-GLOBALD(at_coverage)); /* AT is not performed after recovery or when dirty_bit is zero */ return(1); </pre> |

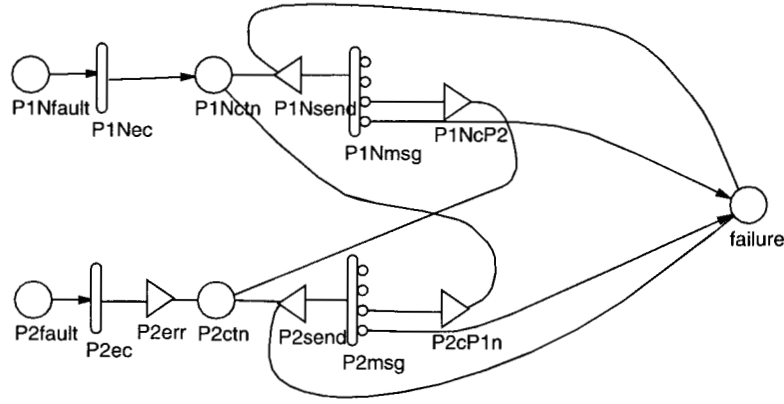


Figure 4: SAN Model for the Baseline System

4.3 Numerical Results

Based on the SAN models developed in the previous section, we analyze the effectiveness of the MDCD protocol using the evaluation tool UltraSAN [13]. In particular, we define reliability as the probability that the system does not deliver erroneous commands to devices (i.e., erroneous external messages) prior to time t . Letting the reliability measures for the system that applies the MDCD protocol and for the baseline system be denoted as R_t^{MDCD} and R_t^{base} , respectively, the numerical solutions of the measures can be obtained by defining a reward rate one for each state of the SAN models in which the marking of the place **failure** equals to one and computing the expected rewards at time t .

As mentioned earlier, the central purpose of the analysis is to validate the effectiveness of the MDCD protocol, in terms of reliability improvement, under the circumstance where the criteria for an ideal execution environment for the protocol are not satisfied. Accordingly, we focus on examining the reliability improvement in an environment where 1) the old software components (corresponding to P_1^{old} and P_2) are not perfectly reliable, 2) the probability that

the error conditions in a contaminated process state are manifested in the messages generated by the corresponding process is less than one, and 3) the coverage of AT is imperfect. Before we proceed to describe the numerical results, we define the following notation:

- μ_{new} Poisson failure rate of a process corresponding to a newly upgraded software version (corresponding to the rate of the timed activity P1Nec).
- μ_{old} Poisson failure rate of a process corresponding to an old software version (corresponding to the rates of the timed activities P1Nec and P2ec).
- p_{s2m} Probability that error conditions in a process state are manifested in a message generated by the corresponding process (corresponding to `prob_s2m`)
- c Coverage of an acceptance test (corresponding to `at_coverage`).
- λ Poisson message sending rate of a process (corresponding to the rates of the timed activities of P1Nmsg, P1Omsg and P2msg).
- p_{ext} Probability that the message a process attempts to send is an external message (corresponding to `prob_ext`).

We first examine the effectiveness of the MDCD protocol by evaluating R_t^{MDCD} and R_t^{base} , for a mission period of 10^4 hours, as a function of μ_{new} . The value assignment for other parameters is shown in Table 4, where all the parameters involving time (durations, rates, etc.) presume that time is quantified in hours. The numerical results are displayed in Figure 5.

Table 4: Parameter Value Assignment

| μ_{old} | p_{s2m} | c | λ | p_{ext} |
|--------------------|------------------|------|-----------|------------------|
| 10^{-8} | 0.9 | 0.95 | 10 | 0.2 |

The curves in Figure 5 show that 1) when μ_{new} is below 10^{-7} , the benefit from applying the MDCD protocol is not appreciable; 2) when μ_{new} becomes 10^{-6} or higher, the reliability improvement becomes increasingly significant; and 3) after μ_{new} reaches 5×10^{-5} , R_t^{base} apparently turns to be unacceptable while R_t^{MDCD} remains reasonable. Thus, based on this particular setting which is rather conservative with respect to the values of p_{s2m} and c , we can observe that the MDCD protocol will offer significant benefit as surmised when the new version is appreciably less reliable than the old version. In other words, the protocol can achieve its goal without requiring the old version of the upgraded software component to be perfectly or extremely reliable. Another interest insight the curves provide to us is that, after μ_{new} reaches 0.001, R_t^{MDCD} not only remains reasonable but also stays steady,

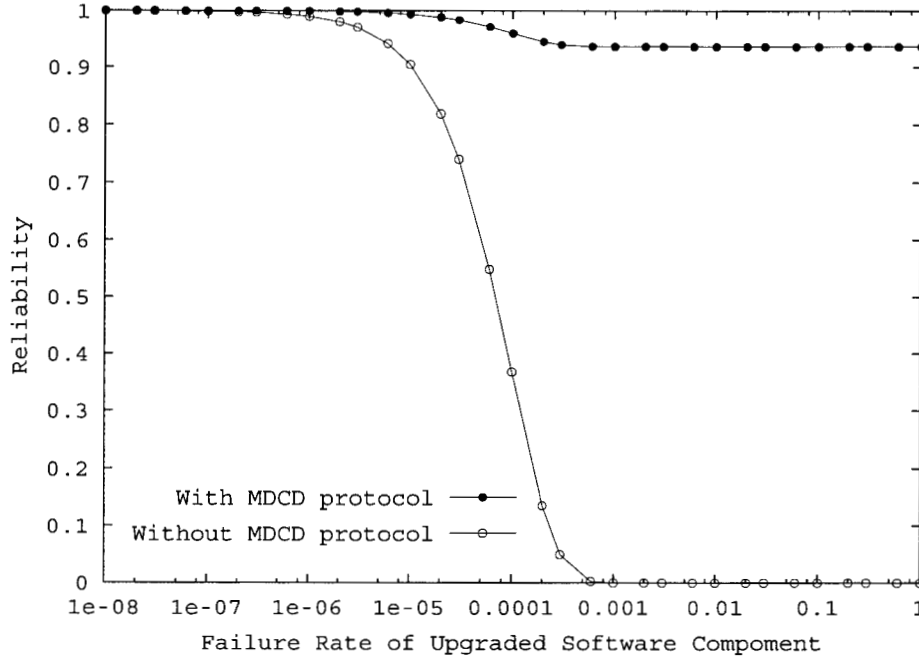


Figure 5: Reliability as a Function of μ_{new}

regardless further increase of the failure rate of the new version. The underlying reason for this desirable result is the following: A higher μ_{new} will lead to a greater likelihood that error recovery will take place at an earlier time (which implies that P_1^{old} will take over P_1^{new} sooner); as a result, μ_{old} will dominate the reliability of the system.

To confirm the above observations from a different perspective, we conduct another analysis that evaluates R_t^{MDCD} and R_t^{base} as a function of μ_{old} . We again use the parameter values shown in Table 4 but fix μ_{new} to 10^{-5} and let μ_{old} become a variable parameter. The numerical results are shown in Figure 6. The observations we get from these results are consistent with those from the previous study. That is, the reliability improvement resulting from the use of the MDCD protocol will be significant if μ_{old} is equal to or less than a value that is an order of magnitude smaller than μ_{new} . On the other hand, the curves reveal that the effectiveness of the protocol increases at a slower pace after μ_{old} reaches 10^{-6} and becomes practically stable after μ_{old} decreases to 10^{-7} . This indicates the following: Although the effectiveness of the protocol is an increasing function of the reliability of the old version in general, it is bounded upper collectively by other system attributes, namely, the coverage of AT, the reliability of the new version, and the likelihood of dormant error conditions that are not manifested in the messages prior to recovery action.

Next we study the effect of AT's coverage on the effectiveness of the protocol. We use again the set of parameter values in Table 4 but fixing μ_{new} and μ_{old} to 10^{-5} and 10^{-8} , re-

spectively, and letting c become a variable parameter. For the sake of illustration, we present the coverage of AT and the evaluation results (R_t^{MDCD} and R_t^{base}) in their complimentary forms in Figure 7. The curves show that so long as AT’s “uncoverage” is less than 0.1 (i.e., c is greater than 0.9), the unreliability reduction (i.e., reliability improvement) from applying the MDCD protocol will be significant.

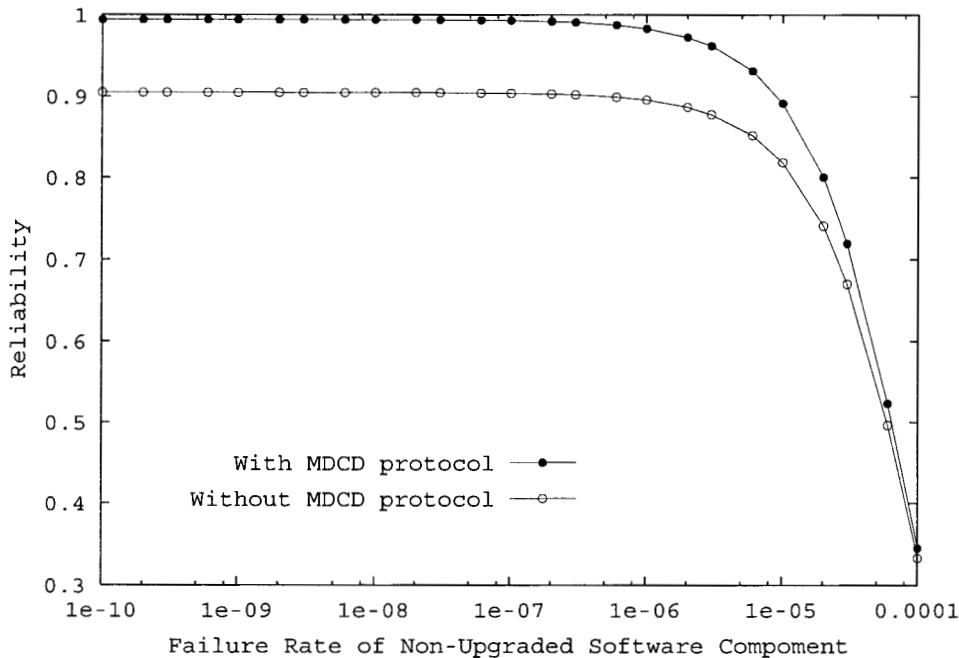


Figure 6: Reliability as a Function of μ_{old}

We also conduct an evaluation to study the effect of p_{s2m} on the effectiveness of the MDCD protocol. Rather surprisingly, reliability improvement from applying the protocol is relatively insensitive to the variations of this parameter. This is indeed a reasonable result because there exist some tradeoffs. Specifically, while a greater value of p_{s2m} tends to reduce the likelihood of dormant error conditions in process states left behind recovery, it amplifies the vulnerability of error contamination among interacting processes (through error condition manifestation in internal messages). In other words, the two types of effects compensate each other, collectively resulting in a negligible amount of impact on the effectiveness of the protocol.

5 Summary and Future Work

We have presented an analysis on the effectiveness of the MDCD protocol, an error containment and recovery protocol for onboard software upgrading. By exploiting inherent

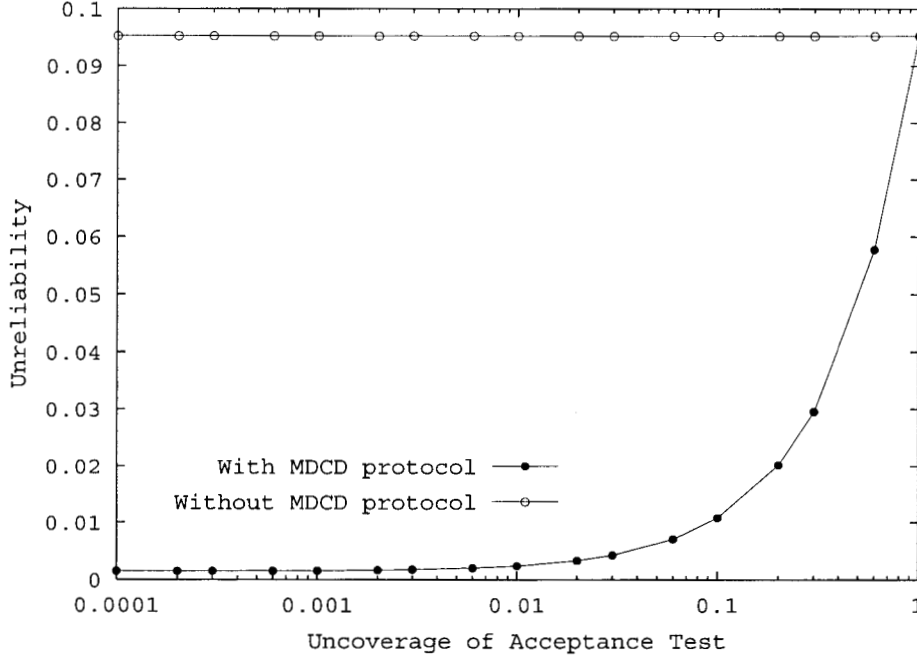


Figure 7: Unreliability as a Function of AT Coverage

system resource redundancies and discriminating interacting software components in the system with respect to our confidence on their reliability, the MDCD protocol achieves its low development cost and low performance cost objective. In particular, the message driven confidence driven nature of the protocol eliminates the need for costly process coordination or atomic action, while guaranteeing the system to reach a consistent global state upon the completion of the rollback or roll-forward actions carried out by individual processes during error recovery.

Aimed at validating the effectiveness of the MDCD protocol with respect to its ability, in a non-ideal execution environment, to enhance system reliability when a software component undergoes onboard upgrading, we conduct a SAN model based analysis. SANs' capability of explicitly representing the interdependencies among system attributes enables us to precisely characterize system behavior resulting from the use of the protocol that are relevant to the reliability assessment. Based on the SAN models, we focus on analyzing the effects of the system attributes, that violate the criteria for the ideal execution environment for the MDCD protocol, on the effectiveness of the protocol. The analysis results confirm the protocol's ability of enhancing reliability for onboard software upgrading in a non-ideal execution environment. Moreover, the model-based analysis provides to us useful insights about the system behavior resulting from the use of the protocol when the criteria for an ideal execution environment are relaxed to various extents, facilitating effective utility of the

protocol.

It is worth to mention that the MDCD protocol described in this paper can be extended and generalized. In particular, the extension and generalization will be aimed at applying the methodology to the distributed systems in which we can discriminate between interacting software components with respect to their reliability. Indeed, a number of factors other than upgrading may result in differing levels of confidence in different software components in a system, for example, we may have better confidence in a software component with lower complexity or higher testability. In other words, software components in a distributed application may be categorized into two groups according to our confidence in their reliability. Analogous to the strategies used by the MDCD protocol, the high confidence group can be exploited to enhance the efficiency of error containment and recovery. We plan to conduct model-based studies to investigate into the feasibility of generalizing the concepts and framework of the MDCD protocol.

References

- [1] L. Alkalai and A. T. Tai, “Long-life deep-space applications,” *IEEE Computer*, vol. 31, pp. 37–38, Apr. 1998.
- [2] J. L. Lions (The Chairman of the Board), *ARIANE 5 Flight 501 Failure*, July 1996. <http://sspg1.bnsc.rl.ac.uk/Share/ISTP/ariane5r.htm>.
- [3] A. Avižienis, “Towards systematic design of fault-tolerant systems,” *IEEE Computer*, vol. 30, pp. 51–58, Apr. 1997.
- [4] J. Rendleman, “MCI WorldCom blames Lucent software for outage,” in *PC Week*, Ziff-Davis, August 16, 1999. <http://www.zdnet.com/pcweek/stories/news/0,4153,2318289,00.html>.
- [5] L. Sha, J. B. Goodenough, and B. Pollak, “Simplex architecture: Meeting the challenges of using COTS in high-reliability systems,” *CrossTalk: The Journal of Defense Software Engineering*, Apr. 1998.
- [6] D. Powell *et al.*, “GUARDS: A generic upgradable architecture for real-time dependable systems,” *IEEE Trans. Parallel and Distributed Systems*, vol. 10, pp. 580–599, June 1999.

- [7] A. T. Tai and K. S. Tso, "On-board maintenance for affordable, evolvable and dependable spaceborne systems," Phase-I Final Technical Report for Contract NAS8-98179, IA Tech, Inc., Los Angeles, CA, Oct. 1998.
- [8] E. N. Elnozahy, D. B. Johnson, and Y.-M. Wang, "A survey of rollback-recovery protocols in message-passing systems," Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Oct. 1996.
- [9] N. Neves and W. K. Fuchs, "Coordinated checkpointing without direct coordination," in *Proceedings of the 3rd IEEE International Computer Performance and Dependability Symposium*, (Durham, NC), pp. 23–31, Sept. 1998.
- [10] A. T. Tai and K. S. Tso, "Verification and validation of the algorithms for guarded software upgrading," Phase-II Interim Technical Progress Report for Contract NAS3-99125, IA Tech, Inc., Los Angeles, CA, Sept. 1999.
- [11] A. T. Tai, K. S. Tso, L. Alkalai, S. N. Chau, and W. H. Sanders, "On low-cost error containment and recovery methods for guarded software upgrading," (Submitted for publication), 1999.
- ✕ [12] J. F. Meyer, A. Movaghar, and W. H. Sanders, "Stochastic activity networks: Structure, behavior, and application," in *Proc. Int'l Workshop on Timed Petri Nets*, (Torino, Italy), pp. 106–115, July 1985.
- ✕ [13] W. H. Sanders, W. D. Obal II, M. A. Qureshi, and F. K. Widjanarko, "The UltraSAN modeling environment," *Performance Evaluation*, vol. 24, no. 1, pp. 89–115, 1995.
- [14] S. N. Chau, L. Alkalai, J. B. Burt, and A. T. Tai, "The design of a fault-tolerant COTS-based bus architecture," in *Proceedings of 1999 Pacific Rim International Symposium on Dependable Computing (PRDC'99)*, (Hong Kong, China), Dec. 1999.

A Error Containment and Recovery Algorithms

```

if (outgoing_message_m_ready) {
    if (external(m)) {
        if (AT(m) == success) {
            // P1new maintains its msg count and conveys it to P2 and P1old for recovery purpose
            msg_count++;
            msg_send(m, null, device);
            // inform P1old and P2 that prior messages are valid
            msg_send("passed_AT", msg_count, P1old);
            msg_send("passed_AT", msg_count, P2);
        } else {
            error_recovery(P1old, P2);
            exit(error);
        }
    } else { // m is an internal message
        msg_count++;
        msg_send(m, msg_count, P2);
    }
}
if (incoming_message_m_arrives) {
    application_msg_reception(m);
}

```

Figure 8: Error Containment Algorithm for P₁^{new}

```

if (outgoing_message_m_ready) {
    msg_count++; // msg_count keeps track of P1old's own messages
    msg_log(m, msg_count); // suppress and log the outgoing message
}
if (incoming_message_m_arrives) {
    if (m.body == "passed_AT") { // P1new or P2 reports a successful AT
        VR1new = m.msg_count; // last valid msg of P1new
        if (dirty_bit == 1) {
            dirty_bit = 0;
            checkpointing(P1old);
        }
    } else { // application-purpose message from P2
        // check the piggybacked dirty bit and own process state
        if (m.dirty_bit == 1 && dirty_bit == 0) {
            checkpointing(P1old);
            dirty_bit = 1;
        }
        application_msg_reception(m);
    }
}

```

Figure 9: Error Containment Algorithm for P₁^{old}

```

if (outgoing_message_m_ready) {
  if (external(m)) {
    if (dirty_bit == 1) {
      if (AT(m) == success) {
        dirty_bit = 0;
        // msg_count of P2 keeps track of msg sequence number of P1new
        msg_send(m, null, device);
        msg_send("passed_AT", msg_count, P1old);
        checkpointing(P2);
      } else {
        error_recovery(P1old, P2);
      }
    } else {
      // outgoing msg from a clean process state, no check needed
      msg_send(m, null, device);
    }
  } else { // internal message
    msg_send(m, null, P1new);
    // piggybacking dirty_bit to msg to P1old to signal possible contamination
    m = append(m, dirty_bit);
    msg_send(m, null, P1old);
  }
}
if (incoming_message_m_arrives) { // must be from P1new
  msg_count = m.msg_count;
  if (m.body == "passed_AT") {
    if (dirty_bit == 1) {
      checkpointing(P2);
      dirty_bit = 0;
    }
  } else {
    if (dirty_bit == 0) { // checkpointing before getting "dirty"
      checkpointing(P2);
      dirty_bit = 1;
    }
  }
  application_msg_reception(m);
}
}

```

Figure 10: Error Containment Algorithm for P₂

```

if (dirty_bit == 1) {
  rollback(most_recent_ckpt);
}
// switch role with P1new and go forward
switch_to_active(VR1new, msg_count);
continue;

```

(a) For P₁^{old}

```

if (dirty_bit == 1) {
  rollback(most_recent_ckpt);
}
// go forward
continue;

```

(b) For P₂

Figure 11: Error Recovery Algorithms